VIETNAM GENERAL CONFEDERATION OF LABOUR

**TON DUC THANG UNIVERSITY**

**FACULTY OF INFORMATION TECHNOLOGY**



**DESIGN PATTERN**

**VISITOR PATTERN AND STATE PATTERN**

*Supervisor*:  **NGUYEN THANH PHUOC**

*Author*: **NGO GIA PHAT – 519H0113**

**LE DAC CHINH – 519H0274**

**BAO HOANG QUI – 519H000**

Class: **19H50202 – 19H50302**

Group**: 2**

**HO CHI MINH CITY, 2023**

VIETNAM GENERAL CONFEDERATION OF LABOUR

**TON DUC THANG UNIVERSITY**

**FACULTY OF INFORMATION TECHNOLOGY**



**DESIGN PATTERN**

**VISITOR PATTERN AND STATE PATTERN**

*Supervisor*:  **NGUYEN THANH PHUOC**

*Author*: **NGO GIA PHAT – 519H0113**

**LE DAC CHINH – 519H0274**

**BAO HOANG QUI – 519H000**

Class: **19H50202 – 19H50302**

Group**: 2**

**HO CHI MINH CITY, 2023**

APPRECIATION LETTER

First and foremost, I would want to express my gratitude to Ton Duc Thang University for providing facilities that include a contemporary library system, a diverse collection of books and materials, and a conducive environment for obtaining and researching knowledge. Thank you very much! Then, I'd want to express my gratitude to one of my professors, Nguyen Thanh Phuoc, who patiently led and instructed me throughout the process of creating the final project.

There are many flaws and restrictions in my final project due to my limited knowledge and reasoning abilities. We look forward to the assistance and contributions of the instructors and teachers to make my final project more comprehensive. Thank you sincerely!

**PROJECT COMPLETED AT TON DUC THANG UNIVERSITY**

I hereby declare that this is the product of my/our own project and under the guidance of Dr. Nguyen Thanh Phuoc;. The research content, results in this topic are honest and not published in any form before. The data in the tables for analysis, comments and evaluation are collected by the author himself from different sources, which are clearly stated in the reference section.

In addition, the project also uses a number of comments, assessments as well as data from other authors, other agencies and organizations, with citations and source annotations.

**If I find any arduous problems, I would like to take full responsibility for the content of my project**. Ton Duc Thang University is not related to the rights and copyright crimes caused by me during the implementation process (if any).

*Ho Chi Minh city, day month year*

*Author*

*(sign and write full name)*

*Ngo Gia Phat*

*Le Dac Chinh*

*Bao Hoang Qui*

TEACHER'S CONFIRMATION AND ASSESSMENT SECTION

**The confirmation part of the instructor**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Ho Chi Minh city, day month year

(sign and write full name)

**The evaluation part of the teacher marks the test**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Ho Chi Minh city, day month year

(sign and write full name)

SUMMARIZE

Visitor Pattern:

The Visitor pattern is suitable for complex object structures that require an operation without modifying the objects themselves. This pattern is often used in compilers and interpreters where the abstract syntax tree represents the program being compiled or interpreted.

The Visitor pattern separates an algorithm from the objects it operates on. It involves defining a visitor class that encapsulates the behavior that needs to be added, and then implementing a visitor interface in the objects to be visited. The visitor class then traverses the object structure and calls the appropriate methods on the objects to perform the desired operation.

State Pattern:

The State pattern is useful in scenarios where an object's behavior depends on its internal state, and the number of states and transitions between them are complex. This pattern is often used in user interfaces where the behavior of a component depends on the state of other components or user inputs.

The State pattern involves defining a state interface and implementing state classes that define the behavior for each state. A context class holds a reference to the current state and delegates requests to the state objects. When the internal state of the context changes, it switches to a different state object that handles subsequent requests.

In summary, the Visitor and State patterns are powerful tools for addressing complex and dynamic scenarios in Java applications. By separating algorithms from objects in the Visitor pattern and handling state transitions in the State pattern, developers can create more flexible and maintainable code.

TABLE CONTENT

[APPRECIATION LETTER i](#_Toc131042582)

[TEACHER'S CONFIRMATION AND ASSESSMENT SECTION iii](#_Toc131042583)

[SUMMARIZE iv](#_Toc131042584)

[TABLE CONTENT 1](#_Toc131042585)

[LIST OF TABLES, PICTURES, GRAPHS 3](#_Toc131042586)

[CHAPTER 1 – VISITOR PATTERN 4](#_Toc131042587)

[1.1 What is Visitor Pattern? 4](#_Toc131042588)

[1.2 Problem 4](#_Toc131042589)

[1.3 Solution 4](#_Toc131042590)

[1.4 Structure 5](#_Toc131042591)

[1.5 Pseudocode 5](#_Toc131042592)

[1.6 Implementation 5](#_Toc131042593)

[1.6.1 Implement in Java 5](#_Toc131042594)

[1.6.2 Description 5](#_Toc131042595)

[1.7 Conclusion 6](#_Toc131042596)

[1.7.1 Applicability 6](#_Toc131042597)

[1.7.2 Pros and Cons 6](#_Toc131042598)

[1.7.3 Relations with Other Patterns 8](#_Toc131042599)

[CHAPTER 2 – STATE PATTERN 8](#_Toc131042600)

[2.1 What is State Pattern? 8](#_Toc131042601)

[2.2 Problem 9](#_Toc131042602)

[2.3 Solution 9](#_Toc131042603)

[2.4 Structure 10](#_Toc131042604)

[2.5 Pseudocode 10](#_Toc131042605)

[2.6 Implementation 10](#_Toc131042606)

[2.6.1 Implement in Java 10](#_Toc131042607)

[2.6.2 Desciption 10](#_Toc131042608)

[2.7 Conclusion 10](#_Toc131042609)

[2.7.1 Applicability 10](#_Toc131042610)

[2.7.2 Pros and Cons 11](#_Toc131042611)

[2.7.3 Relations with Other Patterns 12](#_Toc131042612)

[CHAPTER 3 – REFERENCE 13](#_Toc131042613)

[3.1 Acronym 13](#_Toc131042614)

[3.2 Citation 13](#_Toc131042615)

[3.2.1 References and citations 13](#_Toc131042616)

[3.2.2 Regulations of the Faculty of Information Technology 13](#_Toc131042617)

LIST OF TABLES, PICTURES, GRAPHS

CHAPTER 1 – VISITOR PATTERN

* 1. What is Visitor Pattern?

The Visitor pattern is a design pattern that separates an algorithm from the objects it operates on. It is useful when you have a complex object structure that you want to perform some operation on, but you don't want to modify the objects themselves. The Visitor pattern provides a way to add new operations to existing object structures without changing those structures.

* 1. Problem

A company has developed a large and complex system for managing their customer data. The system includes a variety of different types of objects, such as customers, orders, and invoices. The company wants to be able to generate reports that analyze different aspects of the customer data, such as total sales by region or average order size by customer type. However, adding new reporting functionality to the system is difficult because the objects in the system are not designed to support the required operations.

* 1. Solution

To solve this problem, the company decides to use the Visitor pattern. They create a ReportVisitor class that defines the behavior for generating reports, such as calculating total sales or computing averages. They then implement a Visitor interface in the various objects in the system that need to be analyzed, such as the customer, order, and invoice classes. The Visitor interface includes methods for accepting the ReportVisitor and delegating the required calculations to the visitor.

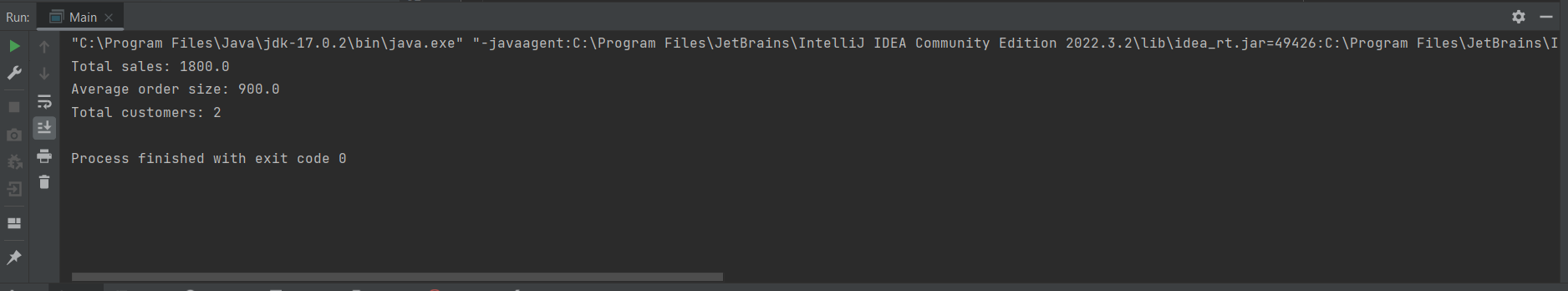
When the company needs to generate a report, they create a new instance of the ReportVisitor class and pass it to the root object in the system. The ReportVisitor traverses the object structure, visiting each relevant object and performing the required calculations. The resulting report can then be displayed to the user or exported to a file.

By using the Visitor pattern, the company is able to add new reporting functionality to their system without modifying the existing object structure. They can easily create new ReportVisitor classes to analyze the data in different ways, making the system more flexible and adaptable to changing business needs.

* 1. Structure
  2. Pseudocode
  3. Implementation
     1. Implement in Java

1. // Visitor interface  
   interface Visitor {  
    void visit(Customer customer);  
    void visit(Order order);  
    void visit(Invoice invoice);  
   }  
     
   // Concrete Visitor implementation for generating reports  
   class ReportVisitor implements Visitor {  
    private double totalSales;  
    private int totalOrders;  
    private int totalCustomers;  
     
    @Override  
    public void visit(Customer customer) {  
    totalCustomers++;  
    }  
     
    @Override  
    public void visit(Order order) {  
    totalOrders++;  
    totalSales += order.getTotalAmount();  
    }  
     
    @Override  
    public void visit(Invoice invoice) {  
    totalSales += invoice.getTotalAmount();  
    }  
     
    // Getters for report data  
    public double getTotalSales() {  
    return totalSales;  
    }  
     
    public double getAverageOrderSize() {  
    if (totalOrders == 0) {  
    return 0;  
    }  
    return totalSales / totalOrders;  
    }  
     
    public int getTotalCustomers() {  
    return totalCustomers;  
    }  
   }  
     
   // Visitable interface  
   interface Visitable {  
    void accept(Visitor visitor);  
   }  
     
   // Concrete Visitable classes  
   class Customer implements Visitable {  
    @Override  
    public void accept(Visitor visitor) {  
    visitor.visit(this);  
    }  
   }  
     
   class Order implements Visitable {  
    private double totalAmount;  
     
    public double getTotalAmount() {  
    return totalAmount;  
    }  
     
    @Override  
    public void accept(Visitor visitor) {  
    visitor.visit(this);  
    }  
   }  
     
   class Invoice implements Visitable {  
    private double totalAmount;  
     
    public double getTotalAmount() {  
    return totalAmount;  
    }  
     
    @Override  
    public void accept(Visitor visitor) {  
    visitor.visit(this);  
    }  
   }  
     
   // Example usage  
   public class Main {  
    public static void main(String[] args) {  
    Customer customer1 = new Customer();  
    Customer customer2 = new Customer();  
    Order order1 = new Order();  
    order1.totalAmount = 100.0;  
    Order order2 = new Order();  
    order2.totalAmount = 200.0;  
    Invoice invoice1 = new Invoice();  
    invoice1.totalAmount = 500.0;  
    Invoice invoice2 = new Invoice();  
    invoice2.totalAmount = 1000.0;  
     
    // Add visitors to visitable objects  
    List<Visitable> visitableObjects = new ArrayList<>();  
    visitableObjects.add(customer1);  
    visitableObjects.add(customer2);  
    visitableObjects.add(order1);  
    visitableObjects.add(order2);  
    visitableObjects.add(invoice1);  
    visitableObjects.add(invoice2);  
     
    // Generate report  
    ReportVisitor reportVisitor = new ReportVisitor();  
    for (Visitable visitableObject : visitableObjects) {  
    visitableObject.accept(reportVisitor);  
    }  
     
    // Display report  
    System.*out*.println("Total sales: " + reportVisitor.getTotalSales());  
    System.*out*.println("Average order size: " + reportVisitor.getAverageOrderSize());  
    System.*out*.println("Total customers: " + reportVisitor.getTotalCustomers());  
    }  
   }

The result:



* + 1. Description

In this example, we have implemented the Visitor pattern to generate reports for a customer data management system. We have defined a **`Visitor`** interface and a **`ReportVisitor`** concrete implementation that defines the behavior for generating reports. We have also defined a **`Visitable`** interface and implemented it in the various objects in the system that need to be analyzed, such as the **`Customer`**, **`Order`**, and **`Invoice`** classes. When the company needs to generate a report, they create a new instance of the **`ReportVisitor`** class and pass it to the root object in the system, which then traverses the object structure and performs the required calculations. Finally, the resulting report is displayed to the user or exported to a file.

By using the Visitor pattern, the company is able to add new reporting functionality to their system without modifying the existing object structure. They can easily create new ReportVisitor classes to analyze the data in different ways, making the system more flexible and adaptable to changing business needs. However, implementing the Visitor pattern can add complexity to the code, as each object in the system must be modified to implement the Visitable interface and accept the Visitor. Additionally, adding new types of objects to the system may require modifying the Visitor interface and all of its implementations.

* 1. Conclusion
     1. Applicability

The Visitor pattern is useful in situations where a complex data structure (such as an object hierarchy) needs to be traversed and processed by multiple unrelated algorithms or operations.

Here are some specific situations where the Visitor pattern can be applied:

* Adding new operations to existing classes without modifying those classes: If a data structure has a large number of classes, adding new operations to all of them can be time-consuming and error-prone. With the Visitor pattern, new operations can be added by creating a new visitor class that implements the necessary behavior for each class.
* Separating algorithms from the objects on which they operate: The Visitor pattern allows algorithms to be defined in separate visitor classes, which can be applied to objects without modifying those objects. This can make the code more modular and easier to maintain.
* Maintaining open-ended sets of operations on a complex object structure: The Visitor pattern allows new visitor classes to be created without modifying existing classes, making it easy to add new operations to a complex object structure.
* Reducing code duplication: If multiple operations require similar logic for visiting an object structure, the Visitor pattern can help reduce code duplication by centralizing the visiting logic in a single place.

Overall, the Visitor pattern is a useful tool for dealing with complex object structures where multiple algorithms or operations need to be applied in a flexible and extensible way.

* + 1. Pros and Cons

Pros of the Visitor pattern include:

* **Separation of concerns:** The Visitor pattern separates algorithms from the objects on which they operate, which can make code more modular and easier to maintain.
* **Extensibility:** The Visitor pattern allows new operations to be added to an object structure without modifying existing classes, which can be useful in large and complex systems.
* **Open/Closed Principle:** The Visitor pattern supports the Open/Closed Principle, which states that software entities should be open for extension but closed for modification. The Visitor pattern allows new behavior to be added without modifying existing code.
* **Code reuse:** The Visitor pattern can help reduce code duplication by centralizing visiting logic in a single place.

Cons of the Visitor pattern include:

* **Complexity:** The Visitor pattern can add complexity to code by introducing new classes and interfaces.
* **Increased code size:** Implementing the Visitor pattern can result in increased code size, which can make the code harder to read and understand.
* **Dependency injection:** In some cases, the Visitor pattern requires dependency injection to work properly, which can add additional complexity to code.
* **Performance overhead:** The Visitor pattern can result in additional method calls and object allocations, which can impact performance in certain situations.

Overall, the Visitor pattern can be a powerful tool for dealing with complex object structures, but it should be used judiciously and with an eye towards balancing the benefits of increased flexibility and extensibility against the potential costs of increased complexity and performance overhead.

* + 1. Relations with Other Patterns

The Visitor pattern can be used in combination with other design patterns to solve complex problems. Some examples of patterns that can be used in conjunction with the Visitor pattern include:

* **Composite pattern:** The Visitor pattern can be used to traverse a composite object structure and perform operations on the individual components.
* **Iterator pattern:** The Visitor pattern can be used to implement iteration over an object structure, allowing the Visitor to visit each element in turn.
* **Strategy pattern:** The Visitor pattern can be used to implement different algorithms that can be applied to an object structure, allowing the Visitor to choose the appropriate algorithm at runtime.
* **Interpreter pattern:** The Visitor pattern can be used to interpret a language or expression by visiting the nodes in an abstract syntax tree.
* **Template Method pattern:** The Visitor pattern can be used as part of a template method, where the Visitor provides a concrete implementation of an abstract method that is called from the template method.

CHAPTER 2 – STATE PATTERN

1. What is State Pattern?

The State pattern is a behavioral design pattern that allows an object to change its behavior when its internal state changes. It is used to model situations where an object's behavior is dependent on its current state and there are different ways the object can transition between states based on external events or triggers. The State pattern is useful for simplifying complex conditional logic and for making code more modular and flexible, as it separates the behavior of an object from its state transitions.

1. Problem

A vending machine company wants to create a new type of vending machine that sells hot and cold drinks. The machine should have different states based on the temperature of the drink being dispensed, such as "hot," "cold," and "room temperature." The company wants the machine to be able to change states based on user input and to provide feedback to the user about the current state.

1. Solution

To solve this problem, the vending machine company can use the State pattern. They can define a State interface that defines the behavior for each state, such as dispensing a drink, checking the temperature, and changing to a new state based on user input. They can then create concrete state classes for each temperature state, such as HotDrinkState, ColdDrinkState, and RoomTempDrinkState.

The vending machine can maintain a reference to the current state object, and delegate behavior to that object based on user input. For example, if a user selects a hot drink, the vending machine can delegate to the HotDrinkState object to dispense the drink and update the state to the ColdDrinkState object. Similarly, if a user selects a cold drink, the vending machine can delegate to the ColdDrinkState object and update the state accordingly.

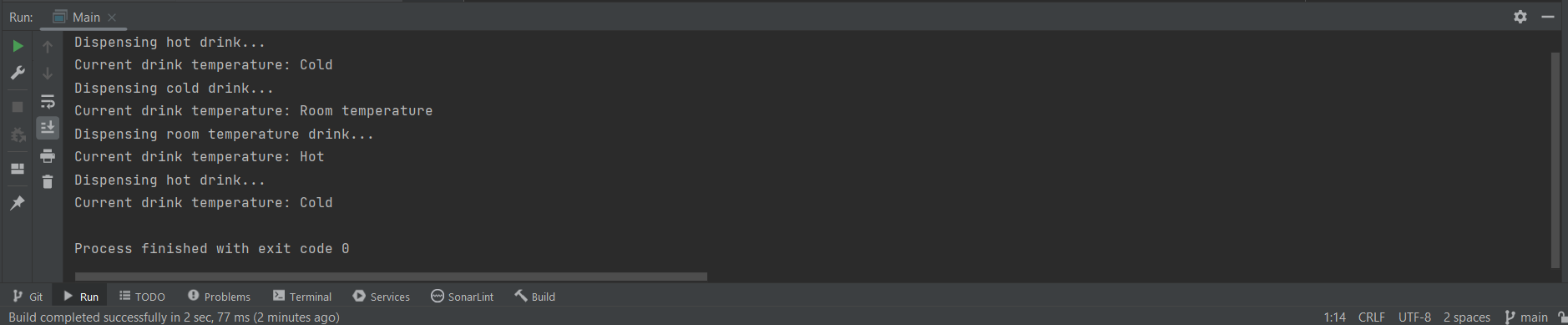
Using the State pattern, the vending machine company can easily add new states and behavior to the machine without modifying the existing code. They can also provide feedback to the user about the current state of the machine, such as displaying "Hot Drink" or "Cold Drink" on a screen. This makes the machine more flexible and user-friendly, improving the overall customer experience.

1. Structure
2. Pseudocode
3. Implementation

2.6.1 Implement in Java

// State interface  
interface State {  
 void dispenseDrink(VendingMachine context);  
 String getTemperature();  
}  
  
// HotDrinkState concrete class  
class HotDrinkState implements State {  
 @Override  
 public void dispenseDrink(VendingMachine context) {  
 System.*out*.println("Dispensing hot drink...");  
 context.setState(new ColdDrinkState());  
 }  
  
 @Override  
 public String getTemperature() {  
 return "Hot";  
 }  
}  
  
// ColdDrinkState concrete class  
class ColdDrinkState implements State {  
 @Override  
 public void dispenseDrink(VendingMachine context) {  
 System.*out*.println("Dispensing cold drink...");  
 context.setState(new RoomTempDrinkState());  
 }  
  
 @Override  
 public String getTemperature() {  
 return "Cold";  
 }  
}  
  
// RoomTempDrinkState concrete class  
class RoomTempDrinkState implements State {  
 @Override  
 public void dispenseDrink(VendingMachine context) {  
 System.*out*.println("Dispensing room temperature drink...");  
 context.setState(new HotDrinkState());  
 }  
  
 @Override  
 public String getTemperature() {  
 return "Room temperature";  
 }  
}  
  
// VendingMachine context class  
class VendingMachine {  
 private State state;  
  
 public VendingMachine() {  
 this.state = new HotDrinkState(); // Default state is hot drink  
 }  
  
 public void setState(State state) {  
 this.state = state;  
 }  
  
 public void dispenseDrink() {  
 state.dispenseDrink(this);  
 System.*out*.println("Current drink temperature: " + state.getTemperature());  
 }  
}  
  
// Example usage  
public class Main {  
 public static void main(String[] args) {  
 VendingMachine vendingMachine = new VendingMachine();  
 vendingMachine.dispenseDrink(); // Dispensing hot drink...  
 vendingMachine.dispenseDrink(); // Dispensing cold drink...  
 vendingMachine.dispenseDrink(); // Dispensing room temperature drink...  
 vendingMachine.dispenseDrink(); // Dispensing hot drink...  
 }  
}

The result:



2.6.2 Desciption

In this implementation, the **`State`** interface defines the behavior for each state, such as **`dispenseDrink`** and **`getTemperature`**. The concrete state classes, **`HotDrinkState`**, **`ColdDrinkState`**, and **`RoomTempDrinkState`**, implement the **`State`** interface and provide the specific behavior for each temperature state.

The **`VendingMachine`** class maintains a reference to the current state object and delegates behavior to that object based on user input. The **`dispenseDrink`** method calls the **`dispenseDrink`** method of the current state object and updates the state of the machine accordingly. It also displays the current temperature of the drink using the **`getTemperature`** method of the current state object.

By using the State pattern, the vending machine company can easily add new temperature states and behavior to the machine without modifying the existing code. They can also provide feedback to the user about the current state of the machine, such as displaying "Hot Drink" or "Cold Drink" on a screen.

1. Conclusion

2.7.1 Applicability

The applicability scenarios for the State pattern:

* When the behavior of an object needs to change dynamically based on its state, and the number of states is relatively small.
* When there are multiple conditional statements based on the state of an object, and the behavior of the object needs to be modularized and managed in a more structured way.
* When a class's behavior changes depending on its internal state, but it is not desirable to expose this state information to the outside world.
* When there are multiple related states and transitions between them, and the code needs to be organized in a way that is easy to maintain and extend.
* When there are complex state machines with a large number of states and transitions, and there is a need to manage these states and transitions in a more structured and maintainable way.

In general, the State pattern is useful when there are multiple related states and transitions between them, and when the behavior of an object needs to change dynamically based on its state. The pattern helps to organize code in a more modular and maintainable way, making it easier to extend and modify as requirements change over time.

2.7.2 Pros and Cons

Pros of State Pattern including:

* **Flexibility:** The State pattern makes it easy to add new states and behaviors to an object without changing the existing code.
* **Separation of concerns:** The pattern separates the behavior of an object into different states, which can make the code easier to understand and maintain.
* **Encapsulation:** Each state encapsulates its own behavior and state, which can help to reduce complexity and improve code organization.
* **Reusability:** The state objects can be reused across different objects or systems, which can save development time and improve code quality.

Cons of State Pattern including:

* **Complexity:** Implementing the State pattern can increase the complexity of the code, especially if there are many states and state transitions.
* **Increased number of classes:** The pattern can result in a larger number of classes, which can make the codebase more difficult to navigate and understand.
* **Potential for tight coupling:** If the state objects have direct references to each other, or if there are many complex state transitions, the code can become tightly coupled and difficult to modify.
* **Performance overhead:** The pattern can add some performance overhead due to the need to constantly check and update the current state of an object.

2.7.3 Relations with Other Patterns

The State pattern has some relationships with other design patterns, including:

* **Strategy Pattern:** The State pattern is similar to the Strategy pattern in that it involves encapsulating behavior in separate objects. However, while the Strategy pattern is used to vary the behavior of an object at runtime, the State pattern is used to vary an object's behavior based on its state.
* **Bridge Pattern:** The State pattern can be used in conjunction with the Bridge pattern to decouple an abstraction from its implementation. The State pattern can represent the various states of the abstraction, while the Bridge pattern can represent the different implementations.
* **Singleton Pattern:** The State pattern can be used in conjunction with the Singleton pattern to ensure that only one instance of a particular state exists at any given time.
* **Flyweight Pattern:** The State pattern can be used in conjunction with the Flyweight pattern to share state objects between multiple context objects, reducing the memory footprint of the application.

Overall, the State pattern can be used in conjunction with other design patterns to improve the flexibility and maintainability of an application.

CHAPTER 3 – REFERENCE

1. Acronym

The abbreviation function cannot be used. Only abbreviate words, phrases or terms that are used repeatedly in the essay. Do not abbreviate long phrases, clauses, or infrequent phrases. If it is necessary to abbreviate words, terms, names of agencies and organizations, etc., they shall be abbreviated after the first writing, accompanied by abbreviations in the citation application. If there are too many abbreviations, there should be a list of abbreviations (in order A, B, C) at the beginning of the essay.

1. Citation
2. References and citations
3. Regulations of the Faculty of Information Technology

**-Plagiarism** is the use of words or ideas of others as their own in academic activities in particular and in creative activities in general. At Hoa Sen University, the following acts are considered plagiarism:

• Copy verbatim 02 (two) sentences in a row without quoting the source in accordance with regulations;

• Copying 03 (three) non-consecutive sentences verbatim without quoting the correct source;

• Rephrase or translate (translate) an entire idea of someone else without citing the correct source;

• Using more than 30% of the content of a final report written by oneself to submit to 2 different classes (same semester or different semester) without the consent of the lecturer;

• Copy part or all of someone else's work.

- When the thesis, project, large assignment is graded, if it is detected plagiarism, it will be immediately scored 0. Students will continue to be disciplined according to the regulations of the University.

**REFERENCES**

* **GitHub:** <https://github.com/ngphat-it-dev/Visitor-Pattern-and-State-Pattern.git>
* **Refactoring.guru:** <https://refactoring.guru/design-patterns/state>
* **Refactoring.guru:** <https://refactoring.guru/design-patterns/visitor>
* **Wikipedia:** <https://vi.wikipedia.org/wiki/Visitor_pattern>
* **Viblo.asia:** <https://viblo.asia/p/tim-hieu-visitor-pattern-qua-vi-du-ZabG9zedvzY6>
* **Medium:** <https://medium.com/@mypascal2000/visitor-pattern-4feb4e004448>
* **Baeldung.com:** <https://www.baeldung.com/java-visitor-pattern>
* **Viblo.asia:** <https://viblo.asia/p/state-design-pattern-tro-thu-dac-luc-cua-developers-3P0lPB9PKox>
* **Gpcoder.com:** <https://gpcoder.com/4785-huong-dan-java-design-pattern-state/>
* **En.wikipedia:** <https://en.wikipedia.org/wiki/State_pattern>
* **Tutorialspoint:** <https://www.tutorialspoint.com/design_pattern/state_pattern.htm>
* **Sourcemaking.com:** <https://sourcemaking.com/design_patterns/state>
* **Topdev.vn:** <https://topdev.vn/blog/huong-dan-java-design-pattern-state/>

**APPENDIX**

This section includes the necessary content to illustrate or support the thesis content such as figures, forms, pictures. . . . if responses to a questionnaire are used, this sample questionnaire must be included in the Appendix in its original form used for the survey and poll; may not be summarized or modified. The sample calculations presented in summary in the forms should also be mentioned in the Appendix of the thesis. Appendices should not be thicker than the main body of the thesis